ARCS - An Architectural Level Communication Driven Simulator

Dave Nellans, Vamshi Krishna Kadaru, Erik Brunvand
School of Computing
University of Utah
Salt Lake City, Utah 84112
{*dnellans, kad, elb*}*@cs.utah.edu*

ABSTRACT

Simulators for digital systems operate at a variety of levels of abstraction varying from detailed analog and switch level modeling of the transistor to cycle based descriptions of entire systems. We propose an even higher level simulator, called ARCS, based on the abstraction of an asynchronous communication event rather than of a clock cycle. Modeling systems at this level allows architectural level exploration of the design space before cycle-level details are available, and also allows the same framework to be used to refine architectural level simulations into more detailed simulations with increasingly fine grained notions of timing. The ARCS simulation framework uses concurrently operating threads in Java with communicating sequential processes (CSP) semantics as a natural expression of communication between concurrent hardware. To avoid synchronization bottlenecks ARCS models time using a communication driven clockwork model which allows for both user configurable runtime viewing of the simulation and post processing of complete simulation timing data.

Categories and Subject Descriptors

C.0 [**General**]: Modeling of computer architecture

General Terms

Algorithms, Design, Experimentation

Keywords

architectural simulation,asynchronous communication, Java

1. INTRODUCTION

Architectural level design exploration should allow a designer to simulate and evaluate system ideas at a high level of abstraction prior to making specific implementation choices. There are many simulators that focus on the detailed transistor- and gate-level implementations of systems [1, 2, 3], and HDL simulators that can simulate both behavioral and structural code in VHDL and Verilog [4, 5], but they model systems at much too fine a level of detail to be easily used for architectural exploration. Others, such as Simplescalar [6], are intended for system level simulation but are often tedious to use because of their fixed simulation granularity based upon the notion of a clock cycle.

At the architectural level the most important considerations are data movement, communication between system pieces, and relative timing between those events. These communication events can be considered asynchronous because at this level of abstraction they are not bound to any particular cycle-level implementation. Due to this, a high level simulation framework that embraces communication as its fundamental simulation unit would be ideal for performing architectural level studies. Once this communication behavior is understood, a specification can be refined into an increasingly detailed implementation.

There are a number of features which we believe are important in an architectural level simulator of this sort. The most important is that the designer must be able to easily define complex high-level behaviors for system pieces and the communication that takes place between them. The simulation must model communication event timing in a way that helps the designer understand the performance impact of system level decisions, and should allow for variable grain size of component descriptions both in terms of behavior and in terms of timing. The simulation should abstract or automate the communication and timing models so that the user is not inundated with details when concentrating on high level design. This trade off between convenience of component abstraction and timing accuracy is crucial to allow rapid prototyping and the refinement of architectural level designs into increasingly detailed implementations. In terms of the design of the simulator itself, it should maintain high performance by avoiding centralized data structures, such as an event queue, and allow its performance to scale with large numbers of simulation components.

Defining these features for an architectural level simulator results in some compromises. The requirement of variable grain size abstraction inherently decreases system timing accuracy. This means that timing estimates based on initial architectural level simulation may need to be refined as more implementation details become available. Abstraction of implementation details such as circuit interconnects does not necessarily provide an easy path for automated synthesis of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-853-9/04/0004 ...$5.00.

circuits; although we note that there are several synthesis systems that are targeted directly at concurrent hardware with CSP-style communication [7, 8, 9, 3].

To support all these features we present ARCS, an asynchronous communication-driven architectural level simulation framework written in Java. ASIM uses CSP-style communications between separate Java threads to implement separate simulated system pieces. The use of Java allows complex behaviors and timing models to be expressed easily. The use of separate threads to model concurrent system pieces, CSP-style communication, and a communication based event timing model means that architectural behavior is effectively supported at a variety of grain sizes. One specific interest we have is in simulating and exploring truly asynchronous systems for which ASIM is ideal [10, 11]. However, because communication is essentially asynchronous at the architectural level this framework is equally well suited for exploring systems that will eventually be implemented as traditional synchronous systems.

2. ARCS FRAMEWORK

We have designed ARCS to fulfill the previously described features resulting in a convenient architectural level simulation framework through which significant performance results can be obtained. A major design choice while writing ARCS was choosing Java as its implementation language. Java was chosen because it is a high level language that allows easy specification of complex behaviors within simulation components while also having native support for concurrency and synchronization; features not often supported in other languages without specialized extensions or requiring significant user expertise. A modern language like Java also encourages the use of ARCS with next generation system designers for whom Java may be a language of choice.

However, Java alone does not provide the needed infrastructure for architectural simulation. Creating a working system that has massive concurrency in any language is not a trivial effort, and Java is no exception. Often implementation details can become overwhelming for even experienced system designers. Because of this, ARCS uses a CSP [12] process algebra as the logical level for modeling processes and their communications. By moving reasoning about these concurrent systems into a formal framework such as CSP a vast body of concurrency and formal methods literature becomes accessible.

CSP style descriptions have been shown to be an effective means of describing concurrent systems [7, 8, 9, 3]. This prior work focused mainly on using CSP as the language from which circuits can be automatically synthesized, although in some cases the CSP-style program can be run to validate functional correctness [13, 3]. These synthesis systems focus on converting the CSP-style descriptions into circuits making implementation details close to the heart of these models; exactly what ARCS is designed to avoid.

ARCS moves in the opposite direction by using the Java object hierarchy to provide increased abstraction details, convenience classes and methods, thereby facilitating rapid development of systems by avoiding implementation details. Also, the use of CSP semantics could allow convenient connections to these hardware synthesis systems if an ARCS description is refined to a small enough grain size.

An additional benefit of abstracting component interconnects is the ability to reason about a simulation asynchronously without concern for the implementation technology. Allowing reasoning about both synchronous systems at a high level, and asynchronous systems at any level, is a major advantage over traditional low-level simulators that focus on the implementation [3, 6].

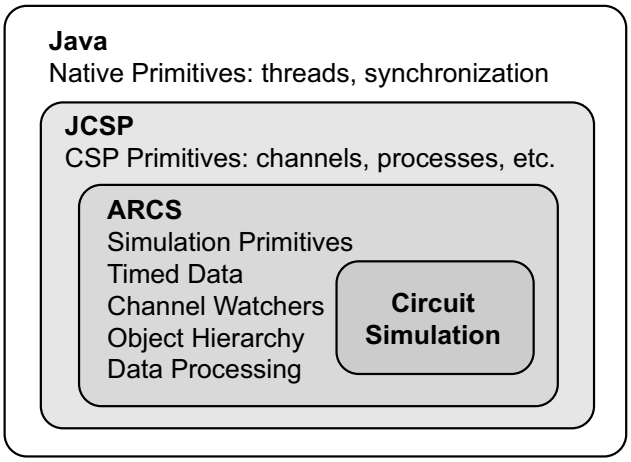

Figure 1: Layering of JCSP and ARCS on Java.

2.1 JCSP

ARCS realizes its CSP model through use of the Java CSP (JCSP) library from University of Kent [14]. The JCSP library is an implementation of the CSP primitives on top of Java's native concurrency objects, threads and semaphores. The use of the JCSP library in ARCS is twofold. First, CSP-level models of concurrency allow for formal proofs of properties using tools such as those from Formal Systems [15]. Secondly, JCSP provides a convenient abstraction of Java's threads and semaphores which are tedious and error prone to work with in quantity. Because ARCS components are instantiated as JCSP/CSP processes it is required that all interprocess communication be channel based; thus there is no explicitly shared memory between threads. This eliminates implementation level race conditions which are a major cause of deadlocks and guarantees correct startup of threads to avoid race conditions due to thread spawning. JCSP has been shown to properly implement CSP semantics [14] using these native constructs. Any simulation using only JCSP processes inherits these correctness properties.

ARCS represents all components as JCSP processes which are interconnected by one or more channels. The granularity and number of components as well as their interconnections are specified by the user while defining the simulation. The user then defines specific component behavior to affect data which flows through the component during simulation. This behavior is not limited allowing complex and simple behaviors as well as not modifying the data at all. By allowing this varying functionality of components the high level nature of Java is leveraged to allow complex component behaviors to be implemented easily.

Running ARCS in the Sun Microsystems JDK version 1.4.2_03-b02 under the Linux 2.6 kernel, every Java thread is a native Linux processes. This allows the JCSP processes to be load balanced across multiple processors within a single workstation. This fundamental feature of ARCS's design helps achieve scalable, user transparent, high perfor-

Function	Description
Read Data	Reads data off an input channel.
Update Local Time	Updates localtime based on later of localtime or latest time attached to data.
Return Local Time	Returns the new localtime to the component that provided data.
Process Data	Perform any necessary data manipulations.
Add Timing Info	Add a new timing record to the data of component name, time received, processing time, and time sending attempted.
Send Data	Block writing to an output channel until data is accepted.
Update Local Time 2	Update localtime based on time returned by data receiver.

Table 1: Execution Loop of ARCS Component

mance simulations. As a simulation's granularity increases to gain timing accuracy the number of Linux processes increases proportionally. This increased number of processes allows the OS scheduler to more efficiently distribute work across multiple processors decreasing the performance loss typically associated with increased simulation accuracy. The JCSP library also contains the functionality to allow CSP channels to communicate remotely over a TCP/IP session [16]. This distributed communication provides simulation scaling which can divide large simulations into discrete sections of CSP processes running on difference machines in separate JVM's but connected through standard CSP channels. Though not user transparent this small inconvenience offers a massive increase in computing resources if desired even across different hardware platforms.

3. TIMING

What separates ARCS from a simple collection of JCSP processes, and turns it into a powerful architectural simulation framework is its timing model. Because components can vary in their functional scope it is necessary to decouple timing information from functionality. Each component must be allowed to take a variable amount of simulation time to complete while also being allowed to complete in any order in real time.

To satisfy these timing requirements ARCS utilizes a communication driven distributed clockworking model. Every component has its own notion of the current time and each data item that passes through a communication channel carries time information within it. A component's local notion of time is updated when it participates in a communication action (either input or output). Because CSP communications are synchronized, input and output components must agree to communicate. When this happens, time information is passed with the data and each component updates its own notion of local time based on this information. A component can also increment its own local time and the most recent time tagged against the current data, to model time spent processing data within that component. JCSP allows n-buffered channels enabling multiple data items to be in flight in FIFO order on a single channel. Additionally, it is trivial to model an n-sized FIFO buffer in ARCS for more accurate simulation.

All simulation data in ARCS must extend the TimedData class. Using TimedData allows components to attach timing information to simulation data as it moves through the various modules via channels. As data moves through a simulation component it is automatically tagged by that component with the component's unique simulation ID, time received, time spent processing, and time it attempted to send that data. When a piece of data has been retired from the simulation it contains a full record of all components it passed through and at what times. This data is then recorded and aggregated by ARCS for post processing. This clockwork timing model allows for most important timing information, including communication delays due to contention, to be recorded without the need for a centralized time queue which is often a major performance bottleneck.

To further understand ARCS's timing model we can examine the execution of a general purpose component within ARCS. Upon initialization each component automatically reads its user defined component time from a file listing based on its fully qualified classname. This defines the simulation processing time the component requires independent of its execution complexity. After initialization has completed it begins the standard communication driven execution loop of read, process, write, shown in Table 1. This generic communication driven component is applicable to a wide variety of hardware subsystems and can be mapped to an implementation in any number of ways. ARCS also allows for variable single component time by permitting the user to modify a processing time function that can return non-static time values based upon the function performed.

One style of hardware component that we are particularly interested in is based on asynchronously communicating components known as micropipelines [17]. Under this model the component begins its execution loop by performing a blocking read, *Read Data*, on its input channels until a channel provides data. The component then updates its stored localtime, *Update Local Time*, to the later of its localtime or the latest time present in the current data's appended timing information. It then writes its new localtime back to the component that provided the data, *Return Local Time*. This critical step allows ARCS to accurately model communication delays due to contention for a resource or implement arbitrary communication delays. Without this write back all communications would appear to occur instantaneously to the sender. The component then processes the received data based on its defined functionality, *Process Data*.

After processing the data, the component is ready to send the data out one of its communication channels. It first adds an additional timing tag, *Add Timing Info*, to the data with its unique component name, time the data was received, time required to process the data, and the time it attempted to send the data. It then blocks, writing to a channel until it the data is accepted, *Send Data*. Finally it receives a second localtime update from the receiver representing the time the communication completed, *Update Local Time 2*. If this time is prior to the sender's localtime then communica-

tion was instantaneous and its localtime remains the same, otherwise communication was delayed for some reason and the sender's localtime must be updated. Once the sender's notion of time is updated, the loop cycles back to *Read Data*.

If components need to implement only basic functionality the simulation designer needs only to define the execution functionality and can rely on ARCS library calls to complete the rest of the steps outlined above. If the component needs increased functionality the user must override a number of functions to implement that custom functionality. In this way ARCS optimizes the common case to provide rapid prototyping while not limiting functionality. In all cases however the distributed clockworking model will encapsulate timing information critical to performance evaluation.

As data is retired from the simulation it is passed into Recorder components which aggregate the timing information included with the data for post processing. This timing information can be either processed within the ARCS framework to retrieve statistical information about execution paths and timing delays among other things or it can be exported to a file for archiving. Java's extensive built-in library of functions, including graphics, make it an attractive language in which to perform post processing of simulation data. Currently ARCS post-processing can only provide basic statistical information or a full output of all timing data from the simulation. More sophisticated graphical analysis tools are being developed to allow more detailed analysis.

3.1 Channelwatchers

Post-processing of simulation data is often not convenient for analysis of concurrent systems. To fulfill the need for a real-time viewing of simulation execution ARCS provides ChannelWatchers, which are a specialized ARCS components. ChannelWatchers may be inserted in an active simulation anywhere a channel exists. This single channel is replaced by two channels with a ChannelWatcher component in between. ChannelWatchers differ from functional simulation components in that they may not affect the simulation or timing data that passes through them although they may examine the data. This restriction cause ChannelWatchers to be completely transparent to the simulation and have zero effect on simulation correctness.

ChannelWatchers are intended to perform actions which notify the simulation designer of an event. These events provide a look into the simulation during execution so information about simulation communications can be gleaned in real-time. Because Channelwatchers are a subset of components, defining custom functionality for Channelwatchers is trivial. We currently have console based ChannelWatcher implementations which output a variety of information upon activation. Work is ongoing to provide large scale real-time visualization of data moving through a simulated system using native Java graphics.

4. TORUS ROUTING EXAMPLE

As a small but illustrative example of ARCS's capabilities we implemented a Torus Routing Chip [18]. The Torus Router is an asynchronous, dimension order, wormhole router that has been designed in several technologies; most recently in 0.6 micron CMOS at the University of Utah. At the architectural level the overall system can be described as a set of processors attached to a mesh-connected set of routers as in Figure 2.

Using ARCS we can describe this system at any number of levels of detail. At the highest level we can describe each processor as a component, and the entire mesh-connected routing network as a single ARCS component. Refining that description results in each Mesh Element in Figure 2 being described as a separate ARCS component. Refining further, ARCS can model each Router element separately where two such elements compose a single Mesh Element.

At this level of detail the Mesh Element is implemented in ARCS as two separate components, each representing a single Router, connected by six independent One2OneChannels. The functionality of each Router is described in just a few lines of Java with the rest of the concurrency, channel, and timing infrastructure inherited from the ARCS framework. Connecting these Mesh Elements into a 4x4 mesh with their associated processors results in four Mesh Element processes, each connected to a data source and sink (the processor model), and to four other Mesh Elements. The processors' writes and reads into the Mesh network are controlled by the main simulation thread which writes and reads data into and out of the simulation as specified by the user.

Using the Cadence toolkit [4] the Torus Routing Chip (TRC) was simulated at the switch level in a 0.6 micron CMOS process. Timings were obtained to model the requirements for a Router element to process both the address words and the data words of the packet. These times represent the processing paths within the transister level model of the Router. These times were then input into ARCS's Router level descriptions as the base processing time. A full 4x4 mesh was simulated in both Cadence and ARCS by routing a 22 word packet containing two address words and 20 data words. This packet was routed from position (0,0) to position (1,1), the longest possible path in a 4x4 mesh. Cadence reported 2148ns required to complete this operation while ARCS reported that only 2067ns were required. ARCS's extrapolated full system timing was accurate within 3.77% of the switch level model. This small discrepency is likely due to both the granularity of the model and interconnect delays which were not modeled by ARCS.

ARCS requires significantly less time to complete its simulation than a transister level simulator. To simulate 100 of the aforementioned packets moving through the TRC, ARCS requires less than a second. Cadence requires approximately 18 seconds to perform this same task. This speedup in simulation execution time when compared to switch level simulations is significant when using ARCS to simulate large systems. It allows much more thorough testing of the system because so much more data can be passed through the simulation in a reasonable amount of time. Using Java as the implementation language also allows convenient, flexible generation and formatting of input data to the system.

ARCS performs full system simulations with more than acceptable accuracy while requiring significantly less design time. It requires less than 100 lines of user written Java to simulate the full 4x4 mesh that Cadence modeled using 16,856 transistors. ARCS also allows much more flexibility in terms of playing "what if" games by modifying the behavior of the routers and their communications. Modifying the Torus Routing Chip to use stateful packet based routing instead of wormhole routing would be trivial using ARCS where as it would require a complete redesign of the Cadence model. Similarly, it is not easy to determine how overall system performance would be impacted by speeding

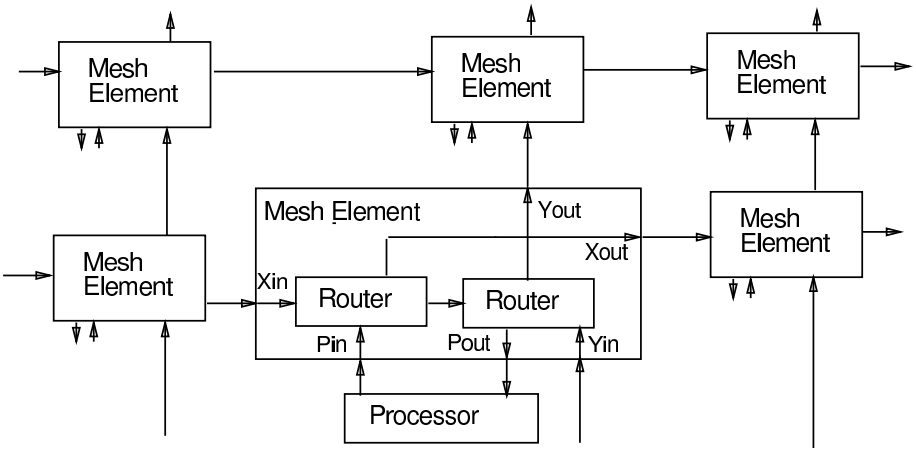

Figure 2: Torus Router

up a single component using switch level modeling. Changing components processing time in ARCS is trivial however. This allows easy experimentation to identify components in which optimization would provide the greatest improvement to total system performance.

5. CONCLUSIONS

This paper has presented a framework called ARCS for performing architectural level exploration in a convenient and efficient manner. ARCS fulfills the key needs of an architectural level simulator by allowing the convenient description of complex functionality using Java, the abstraction of communication as the fundamental unit of simulation, and the separation of component timing and functionality through the use of a distributed clockworking timing model. Because communication is essentially asynchronous at the architectural level this framework is equally well suited for exploring systems that will eventually be implemented as asynchronous systems or as traditional synchronous systems.

ARCS allows rapid prototyping of systems through variable component granularity and by offering support for its timing model through a set of predefined Java classes. By realizing true concurrency during simulation through the JCSP library it is highly scalable without significant synchronization bottlenecks. This allows for a single simulation to be distributed over multiple machines to increase simulation performance if necessary. ARCS's layering on top of the JCSP library encourages leverage of a large body of work on CSP semantics and on synthesis of CSP based descriptions. It also encourages the use of formal verification during the development cycle of systems.

6. REFERENCES

[1] Meta-Software Inc. Hspice user's manual, June 1987.

[2] A. Salz and M. Horowitz. Irsim: an incremental mos switch-level simulator. In *Proceedings of the 26th ACM/IEEE design automation conference*, pages 173–178. ACM Press, 1989.

[3] A. Bardsley and D. Edwards. Compiling the language Balsa to delay-insensitive hardware. In C. D. Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL)*, pages 89–91, April 1997.

[4] Cadence Design Systems. Verilog-xl reference manual, December 1994.

[5] Mentor Graphics. Modelsim se user's manual, 2001.

[6] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[7] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[8] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. ICCAD*, pages 262–265. IEEE Computer Society Press, November 1989.

[9] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[10] W. F. Richardson and E. Brunvand. An architecture for a self-timed decoupled computer. In *Int. Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[11] W. F. Richardson and E. Brunvand. Architectural considerations for a self-timed decoupled processor. *IEE Proceedings, Computers and Digital Techniques*, 143(5):251–257, September 1996.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] E. K. Brunvand and M. Starkey. An integrated environment for the design and simulation of self timed systems. In A. Halaas and P. B. Denyer, editors, *VLSI 91*, page 4a.2. IFIP, August 1991.

[14] P.H.Welch and J.M.R.Martin. Formal Analysis of Concurrent Java Systems. In P.H.Welch and A.W.P.Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 275–301. WoTUG, IOS Press (Amsterdam), September 2000.

[15] Ltd. Formal Systems (Europe). Failures divergence renement: Fdr2 user manual, October 1997.

[16] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.

[17] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[18] E. Brunvand, M. Michell, and K. Smith. A comparison of self-timed design using FPGA, CMOS, and GaAs technologies. In *Proc. ICCD*, pages 76–80. IEEE Computer Society Press, October 1992.